1-1-2002

Blocking API calls for security

James Douglas Truckenmiller
Iowa State University

Blocking API calls for security

by

James Douglas Truckenmiller

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Doug Jacobson, Major Professor
James Davis
Daniel Norris

Iowa State University

Ames, Iowa

2002

Graduate College
Iowa State University

This is to certify that the master's thesis of

James Douglas Truckenmiller

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy

ABSTRACT

The typical user feels comfortable with just having anti-virus software running on their computers. This solution works well if viruses are known and the virus databases are updated frequently. The virus writing community has responded by writing software that mutates the viruses so that these viruses are undetectable to the anti-virus software. Attackers know that anti-virus solutions rely on signatures and if the attacker mutates the virus, the virus will not be detected. Virus writers mutate their product by using software called packers, compressors, and binders. Take an older virus, pass it through one of these mutating programs and now the attacker has a new variant of the old virus that will pass through anti-virus software. A simple example of this type of problem in anti-virus software is the MiniZip worm. A packer called NeoLite was used to create the MiniZip worm, which was a compressed version of the ExploreZip worm. This new variant of the ExploreZip worm spread rapidly even though the virus was well known. This new variant went completely undetected from the existing anti-virus software.

In response to these new threats, a more proactive strategy to counter malicious code must be developed. This thesis focuses on using proactive monitoring techniques to identify code that has malicious intent and to block these operations. To achieve this, the thesis explores digital DNA, system resource monitoring and API monitoring. API monitoring was selected as the method of choice for determining malicious intent. This work discusses different API monitoring technologies, to include: proxy DLL, patching, and binary rewriting. With binary rewriting technology, the author was able to develop a software solution to counter malicious code. As a proof of concept, the author will demonstrate his security product monitoring, and block the one of the most costly viruses to date, the "I love you Virus."

TABLE OF CONTENTS

CHAPTER 1. BACKGROUND **1**

DEFINITIONS 1
ANTI-VIRUS PROBLEMS 2
BEYOND ANTI-VIRUS 3
THESIS GOALS 5

CHAPTER 2. CAPTURING THE DATA **7**

FORMAL DEFINITION 7
MONITORING TECHNIQUES FOR API 8
Proxy DLL 9
Patching 11
PE 11
Code Rewriting 13
Detours Library 13
Detours Efficiency 17
Running in Process 18

CHAPTER 3. VIRUS ATTRIBUTES **20**

ESCALATING PRIVILEGES 20
Services 20
Local buffer Overflows 20
ENUMERATION 21
REGISTRY 21
HOW THE REGISTRY WORKS 21
WRITING FILES 22
Files and Directories Viruses like to Change 23
EMAIL ACCESS 24
INTERNET ACCESS 24

CHAPTER 4. IMPLEMENTATION **26**

REGISTRY API CALLS 27
FILE SYSTEM CALLS 30
File API Calls 30
OVERVIEW OF THE SECURITY PROGRAM 32
Difference between executables and scripts 33
TECHNICAL DESCRIPTION 33
Named Pipe 34
Data Types 34
Threads 35
Determining API Calls 35

CHAPTER 5. CASE STUDY, THE 'I LOVE YOU' VIRUS **37**

BACKGROUND ON 'I LOVE YOU' 37
IN DETAIL 38
REPORT CARD 39

CHAPTER 6. POSSIBLE ATTACKS **40**

NATIVE API 40

SELF-MODIFYING CODE 43
INDIRECT JUMPS 43
CHAPTER 7. CONCLUSION 45
FUTURE WORK 45
FINAL THOUGHTS 46

LIST OF FIGURES

Figure 1. A Simple Model (adapted from Pandey 1998) 7
Figure 2. An Expanded Model 8
Figure 3. Proxy DLL Operation 10
Figure 4. PE Header (adapted from Pietrek, 1994) 12
Figure 5. Binary Rewriting (adopted from Hunt and Brubacher, 1999) 15
Figure 6. Detours Binary Rewriting (adapted from Hunt and Brubacher, 1999) 16
Figure 7. Detours Compared (adapted from Hunt and Brubacher, 1999) 17
Figure 8. Timing Degradation 18
Figure 9. The Front End 26
Figure 10. Selecting the Registry File 29
Figure 11. Menu Options 32
Figure 12. Improved Security Program Model 41
Figure 13. Execution of a Win32 API Call (adapted from Russinovich, 1998) 42

ACKNOWLEDGMENTS

The author wishes to thank his major professor Doug Jacobson and well as the other members of the Program of Study committee. Thanks goes to Amber Nightengale and Patrick Glennon for helping the author edit this paper. Special thanks go to Doug and Linda Truckenmiller for the moral support need to complete a project like this.

Chapter 1. BACKGROUND

Malicious code has plagued the computer industry for as long as computers have existed. Great steps have been made in closing systems and adding security to products, but writers of malicious code have found new and innovative ways to exploit operating systems. The information assurance community has answered with various types of anti-virus software. This solution works well if viruses are known and the virus database is updated frequently. The virus writing community has responded by writing software that transforms the viruses so that they are undetectable to the virus software.

This thesis proposes to show the inadequacies of anti-virus software, supporting the need for a new model to create an additional layer of defense to guard against malicious code. For purposes covered by this thesis, malicious code is defined as the following three major threats: viruses, Trojan horses and worms.

Definitions

To outline a definition for each of these threats, a virus, not termed as a technical exploit, simply attacks the user's action. As a user executes an attachment containing a virus, it launches a pre-programmed sequence of events to attach and infect the user's system. A worm is defined by the propagation of a piece of code across networks. It usually does not attach itself to other programs, but rather is a technical fault in the operating system. An example of this is "Code Red", which exploited a buffer overflow in Microsoft's web server, enabling it to maliciously attack other systems. This required no user intervention. As long as a system was running a web server, it was vulnerable. Trojans are defined by a piece of code that on the outside performs expected

operations, while at the same time is executing additional malicious actions against one system. Software of this nature typically installs a backdoor into the user's system, leaving it open to further manipulation by the attacker. Although there are many more types of malicious code, I will focus on these three.

Anti-Virus Problems

Anti-virus is not infallible. It cannot protect against a computer virus that is new to the security scene. It is intuitive that if the virus signature is not in the database, the virus scanner will not detect it and its malicious code could slip through the cracks of anti-virus protection. Virus writers have also been able to sneak known viruses through anti-virus protection. Attackers know that anti-virus solutions rely on signatures and if the attacker mutates the virus, the virus will not be detected. Virus writers mutate their viruses by using software called packers, compressors, and binders. Take an older virus, pass it through one of these mutating programs and now the attacker has a new variant of the old virus that will pass through anti-virus software. One would think that anti-virus companies would obtain viruses and then use these mutating programs to create signatures for all possible forms of the virus. It is unfeasible for anti-virus companies to put all possible signatures into a database because the combinations are endless. Vincent Weafer, Director for Symantec Anti-Virus Research said, "If anti-virus programs included computer code to read every type of file compression software, the scanning programs would be huge and incredibly slow. The anti-virus companies worry that this would frustrate users and cause them to disable the program." A simple example of these types of problems in anti-virus software is the MiniZip worm. A packer called NeoLite was used to create the MiniZip worm, which is a compressed

version of the ExploreZip worm. This new variant of the ExploreZip worm spread rapidly and went completely undetected from anti-virus software.

Another problem posed to the writers of anti-virus software is the increased propagation speed of new viruses. In earlier times, anti-virus companies could detect a virus, spend a couple of days disassembling it, release a patch and consider their users protected. In today's times, however, viruses spread in hours leaving anti-virus companies no time to release a patch before their customers get infected. Mike Hill, a former Dr Solomon's executive, now an independent technology marketing consultant said, "The fundamental problem is that anti-virus software is reactive. We have a situation now where a virus could have done 80 percent of its damage before a fix has been written - we're shutting the stable door after the horse has bolted."

Beyond Anti-Virus

Now that we have seen the failures of anti-virus software, proactive security companies have taken additional steps to secure their clients' computer systems. These companies have implemented strategies to analyze code in an attempt to determine if the software is malicious. These methods do not use virus definitions, instead relying on the fact that viruses perform uncommon system operations to achieve their malicious intent.

If one could define and monitor these events in the operating system, then it would be possible for a piece of security software to stop malicious code that is new to the scene and therefore previously undefined.

This is a proactive solution. No longer does a computer need to have a window of vulnerability lying between the time the malicious code is released and the point at which the anti-

virus software develops an update. Foundations for this proactive system might include: Digital DNA, system resources, and system calls.

Digital DNA is a relatively new idea. It is the belief that most new viruses inherit source code from older viruses. If one could capture the DNA of the old viruses, it may be possible to scan for new viruses using old DNA. In this way, a software package could identify unknown viruses. The procedure is analogous to the testing of DNA in humans. In much the same way, human offspring can be identified as related to his or her parents through DNA matching. This is a promising technology and there is little doubt that it will be thoroughly investigated in the future. Unfortunately, the mechanics of digital DNA are extremely complex. A solution of this magnitude would require substantial investigation, and therefore is ill suited to exploration in this thesis.

System Resources are metrics on a system that allow the user to determine allocations of a computer's resources. Malicious code needs computer resources to spread, infect, and run. The system can monitor these resources with precision to determine if malicious code is running in the background. They can also employ past metrics as a baseline to test if the machine is using an inordinate amount of resources, which would indicate a possible virus infection.

This method has proven to be unreliable in determining virus activity. A cleverly written virus would not fluctuate a system's resources, but instead demand only a very small amount. Some malicious code writers have refined code to the point that it is nearly impossible to detect with this method. It is true, however, that some viruses are noisy to the system and can still be

detected with this method. This method was not chosen based on the fact that the author feels this method is too easily beaten and not accurate to the extent demanded by this thesis.

The final method investigated uses API calls to differentiate the intent in code. This method differentiates constructive from detrimental code by peering into parameters sent to the application protocol interface. This is the method chosen for this thesis and the author believes it to be superior to all other methods discussed. Since this method was chosen to be the building block of a new security program, its pros and cons will be discussed later in this thesis

Through additional research, it was discovered that many companies implement these strategies; however, not one software product free to the public was available. There are software packages from companies like Network Associates and Finjan available for large sums of money, which protect consumer systems from unknown threats. Not only are these software packages expensive, the companies themselves are unwilling to discuss in detail their technical solutions for proactive code analyzation. This lack of free software justifies the need to further investigate avenues to bring this technology to the public for free.

Thesis Goals

This thesis is centered around the creation of a security program that attempts to spy on Application Protocol Interface (API) calls to search out malicious intent. As stated above, analyses of parameters that are passed to the API, coupled with pattern matching would safeguard against new malicious code attacks. Although this is a proactive solution in this arena, it is by no means the ultimate solution. It is simply an additional safeguard that attackers must circumvent. This security project raises the expertise needed by a code writer to create malicious code that actually

does damage. However, with more layers added to the security model, the more expensive security becomes for the consumer. There is also the increased probability that false positives are introduced into the model.

The proactive security program will focus on analyzing code downloaded by the user. This downloaded code could be an email attachment as well as an executable. Java and active X technologies are not supported. Downloaded code was chosen because it represents a major problem for the typical user. Many of the fastest spreading and destructive viruses were spread through email as attachments. This security program attempts to plug this hole and allow users to open attachments in a safe environment.

The security program will be written for the Microsoft Windows XP environment. This was chosen because at the time of writing this thesis, Windows XP is the latest operating system from Microsoft. This will give the security program at least a two-year life, coinciding with Microsoft's projected release of another operating system. The security program will be developed in Microsoft's Visual Studio .Net environment. It was the author's desire to uses the latest development platform as well as operating system for this thesis.

Chapter 2. CAPTURING THE DATA

Formal Definition

Figure 1 shows that a mobile code C, can be downloaded or migrated to H_1. On

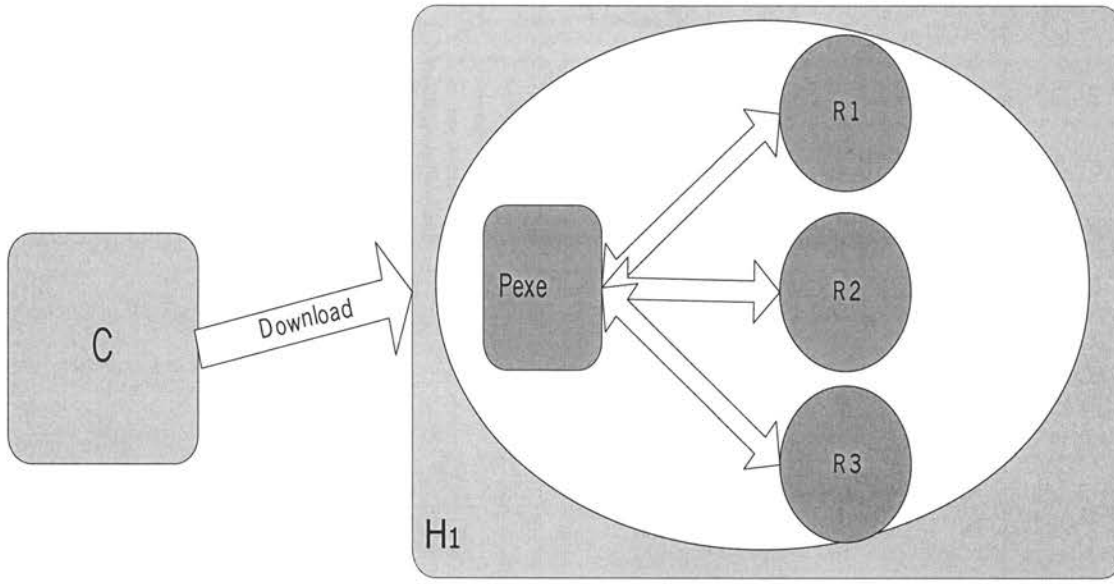

Figure 1. A Simple Model (adapted from Pandey 1998)

H_1, C transforms at runtime to P_{exe}. P_{exe} is linked to H_1 system resources R_1, R_2 and R_3 by H_1 operating system or more specifically its linker. The operating system then creates an environment for P_{exe} to execute in. This is the standard access model which most security products are based on. Security with this model is designed to enforce access control from P_{exe} to the resources R_1 through R_3. This picture is simple but incomplete for this thesis. Figure 2 shows a more complete picture. This model introduces a layer between the resources and P_{exe}, which is labeled I_1. For this thesis, I_1 is defined as the amalgamation of the user API, native API, and system kernel. The interiors of the I_1 layer will be important to the reader, and will be presented in

chapter 6. With these layers defined one can now define the exact nature of this thesis. A security program must place user-defined limits on P_{exe} from accessing R_1, R_2, and R_3 through interface I_1.

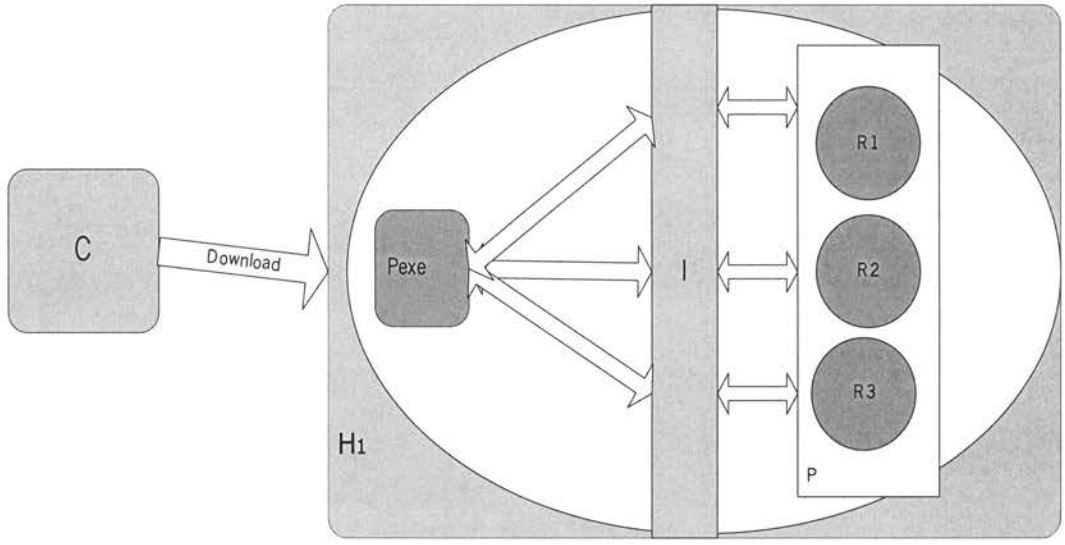

Figure 2. An Expanded Model

From this abstraction, common names will be placed on these formal definitions so that it is specific to the Windows environment. P_{exe} will be referred to as the application, downloaded code and executable. The resources will be called files, registry, and input and output operations. I_1 will be referred to as the Win32 API. Now that the definitions are in place, a look at the technologies available for intercepting API calls is examined.

Monitoring Techniques for API

The following is a list of monitoring (spying) techniques. It is a list of known methods to wedge a monitoring program between the application and the operating system resources. The pros and cons of each technique will be explored.

Proxy DLL

The first and by far the simplest way to spy on an application is to create a proxy DLL (Kaplin, 2000). This proxy DLL must reside in the same directory as the application upon which one wishes to spy. The proxy DLL is to be made up of code stubs that call the real DLL that the application will import functions from. When a user double clicks, or executes a program, Windows uses two functions internally to load and execute the application. LoadModule is used for 16 bit applications whereas Createprocess is used for 32 bit applications. The executed application relies on functions exported by Windows DLLs. Since one requires the operating system to call the proxy DLL before the Windows DLL, the search path of these loading functions must first be determined.

Both of these functions will use a predefined search path. First, the loader will look in the directory in which the executable resides. If the loader does not find its desired DLL, it then looks to the current directory of the calling process. From there, the loader looks at the system directory to locate the specified DLL. By first looking at the executable directory, the proxy DLL is initiated instead of the system DLL. For example, if our goal was to monitor the wininet.dll which has API calls to access services like http, ftp and other related functions, we would create a proxy DLL called wininet.dll and write stubs to call the wininet.dll that resides in the system32 directory. This method is transparent to the executable running (Kaplin, 2000).

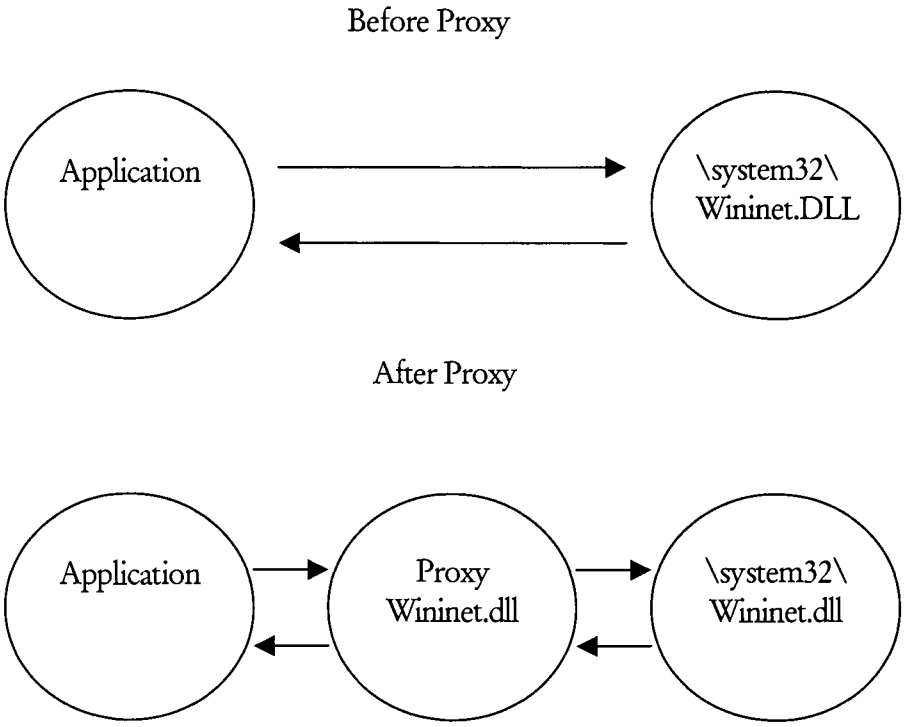

Figure 3. Proxy DLL Operation

One might wonder if Windows will function properly if there are two DLLs with the same name in the same memory space. The answer is yes. Windows will function correctly, as it uses the full system path to identify the DLL. To simplify this, as long as the DLLs are in different directories they can have the same name.

The downside to this method is two fold. The first problem is that most DLLs export hundreds of functions. It would be very tedious to create stubs for all of them (Kaplin, 2000). The second and more pressing problem lies in the fact that Microsoft has many undocumented functions. As the programmer does not know the formal definition of these functions, it becomes

very difficult for him or her to write stubs to these functions. If these undocumented functions are not included in the proxy DLL, then this method loses the transparency to the executable, which is unacceptable. For example, the calculator function of Microsoft Windows might use undocumented functions. If the security program is used to monitor the calculator's system calls, the security program will inhibit the calculator's ability to run. Even if the calculator contained no malicious code, its program would terminate.

Patching

Another technique for API spying is patching. To understand patching, an explanation of the Windows 32 portable execution file format is in order.

PE

Portable execution (PE) is a standard way for executable files to be organized. The format is portable because it does not depend on the processor being used in the system. A PE file can be run on a MIPS, Alpha, or an Intel processor as long as a Microsoft operating system is used. The most prudent sections of the PE file are the import section and the .text and .idata sections. Here is a look at the PE:

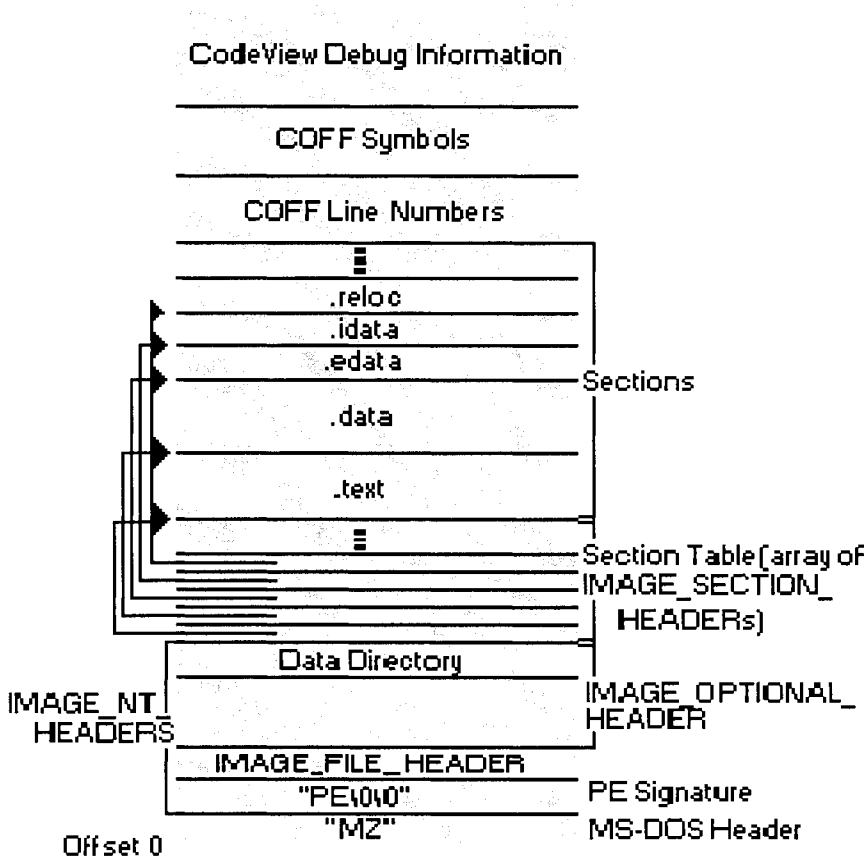

Figure 4. PE Header (adapted from Pietrek, 1994)

As a program executes a call to a DLL, it does not directly jump to the dynamically linked library. The execution flow is instead transferred through an indirect function in the .idata section. In assembly language, it appears as follows:

JMP DWORD PTR [xxxxxxxx]

The reason for this transfer was speculated by Mark Pietrek in the 1994 *MSDN Library.*

This .idata section DWORD contains the real address of the operating system function entry point. After thinking about this for a while, I came to understand why DLL calls are implemented this way. By funneling all calls to a given DLL function through one

> location, the loader doesn't need to patch every instruction that calls a DLL. All the PE loader has to do is put the correct address of the target function into the DWORD in the .idata section. No call instructions need to be patched. .... If the segment calls a given DLL function 20 times, the loader must write the address of that function 20 times into the segment (Pietrick, 1994).

Having the loader change the address 20 times would slow down the loading process. As Microsoft wants their applications to load quickly, they have implemented a single spot for changing the address of API calls.

Since the import address table is a writable, one would let the loader assign this area with the correct function address, save it, and then redirect it to point to a spying function. In these functions one would then point to the address copied from the import address table after it is modified. This would make hijacking of the import address table transparent to the executable. This is one of the most common ways to intercept API calls (Pietrick, 1994). It is efficient because all that is needed is a change to the import table. The downfall of this method is that the code of the program can dynamically call a DLL without going through the import table. Examples of this are the following API calls:

Loadlibrary – can be used to load a new DLL that is not in the import table

GetProcAddress – Loads a function address given a name of the function

Code Rewriting

Detours Library

The Microsoft research team has scrutinized the process of capturing and extending the functionality of Windows systems calls. They put together the detours library, which has an

extensive amount of functions to be used to facilitate API spying. This package is the starting point for the research code introduced by this thesis. It seemed well documented and contained a plethora of examples to aid in familiarization of this package. The detours package uses binary rewriting technology to facilitate its needs. It is superior to other aforementioned technologies (Hunt, 1999).

Binary rewriting in not done to the program on the disk, but it is applied dynamically at runtime. Detours looks for function calls in the binary file and replaces the first five bytes with an absolute jump. This jump points to a detours function in which a programmer can insert custom code. After the custom code is run, control is then transferred to a trampoline function. This trampoline function holds the original code from the function that was detoured. The detour libraries allow one to completely replace a function with one of their own without calling the original. Applying the principles of this security program, one can ultimately place monitoring code in the detours function and then call the trampoline function to make the spying completely transparent. Figure 5 is an illustration of what was just described.

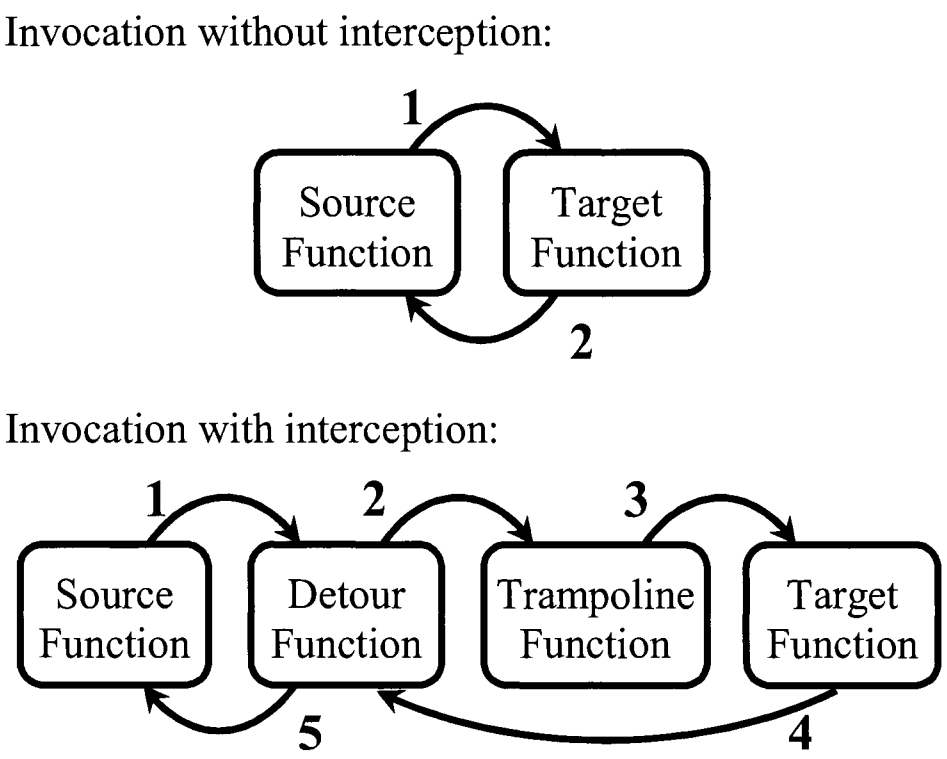

Figure 5. Binary Rewriting (adopted from Hunt and Brubacher, 1999)

Figure 6., adapted from the detours literature, shows the process at the assembly language level. As illustrated in figure 6, the target function is replaced with a jump to a detour function. The detour function can call the trampoline function that restores control to the application being monitored.

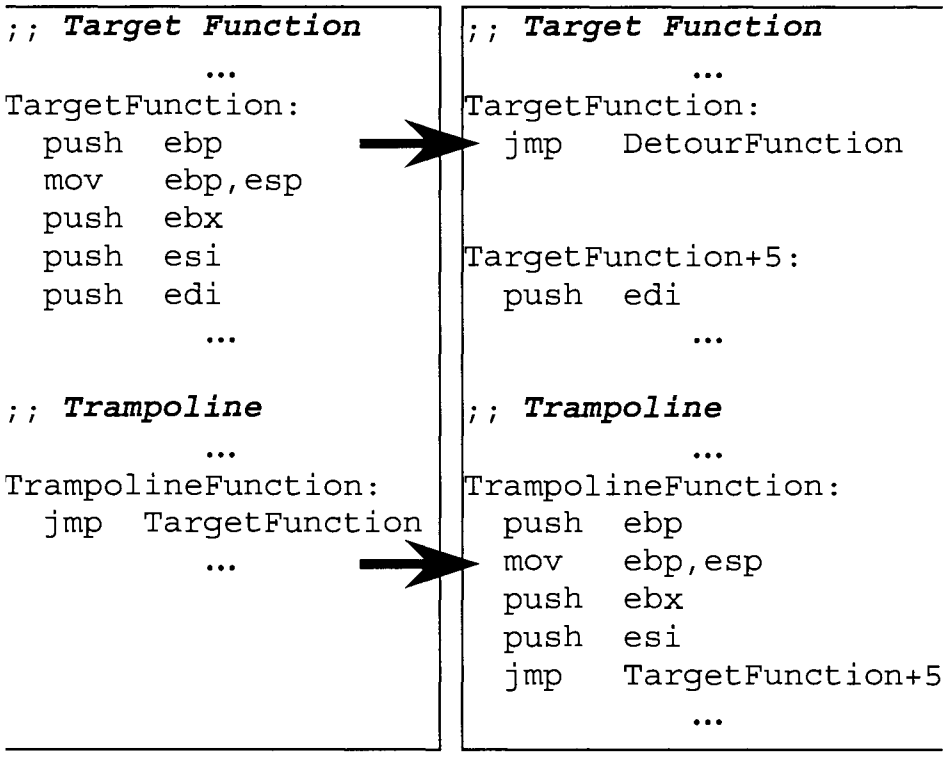

Figure 6. Detours Binary Rewriting (adapted from Hunt and Brubacher, 1999)

One problem with this type of implementation is that a disassembler is needed to properly calculate the instructions to copy. It is necessary to know how many bytes to copy as well as the boundaries of the function calls. Writing a dissasembler is not a trivial task. It would be possible to implement this rewriting strategy; however, the hours necessary to construct this type of program would be in the hundreds. The decision was made not to reinvent the wheel, and the author used the detours library. The time that would have been spent on writing a binary rewriting platform was instead applied to extending the functionality of the detours library.

Detours Efficiency

The detours library is efficient. In the literature distributed with the detours library, it is compared to other techniques explored within this thesis. As illustrated in figure 7, the detours library has very little overhead with an empty function call. For instance, on a call to CoCreateInstance, which creates one object on the local system, the detours library performed respectably when placed against its competitors. Breakpoint trapping was not discussed in this thesis as a viable option for API interception because of its slow performance of Microsoft's debugging system.

Interception Technique	**Intercepted Function**	
	Empty Function	`CoCreate-Instance`
Direct	0.113μs	14.836μs
Call Replacement	0.143μs	15.193μs
DLL Redirection	0.143μs	15.193μs
Detours Library	0.145μs	15.194μs
Breakpoint Trap	229.564μs	265.851μs

Figure 7. Detours Compared (adapted from Hunt and Brubacher, 1999)

The author has noticed degradation in performance when a large number of APIs are monitored with detours. This situation becomes extremely slow when one outputs the API used and the program crashes. Figure 7 shows a table of the times measured by the authors on the calculator program that comes with Windows XP. As one can see, monitoring every API really reduces the transparency of the detours library.

Run	Time
Calc.exe	150 milliseconds
Calc.exe Full Monitoring	2357 milliseconds
Calc.exe Full Output	Crashed repeatedly

Figure 8. Timing Degradation

Running in Process

The API spying code must be in the process of the binary that one would want to monitor. There are multiple ways of achieving this:

1. Change this registry key to include the DLL that one wants:
    - HKEY_LOCAL_MACHINE\Software\Microsoft\WindowsNT\CurrentVersion\Windows\AppInit_Dlls
2. Create a device driver and inject it when CreateProcess is called
3. Modify the PE header to add a DLL with custom code

The third method was chosen. This method modifies the PE header to add a custom DLL. Option two was discarded because the author has little knowledge of the workings of a device driver. Option one, which was to change the registry key, will not work for all programs

that can be run. Changing the registry key links any DLL to the user32.dll. User32.dll is a core DLL upon which Windows programmers heavily rely.

Most Windows graphical programs include user32.dll. Unfortunately most of the DOS programs do not. Option one is the easiest to implement, however it would make this project less effective as it excludes DOS based programs. The detours library has functions to write into the PE header. These were used to change the header to incorporate a new DLL into the process's memory.

The PE header can be changed either dynamically or statically. As far as testing has gone, it was discovered that there was no benefit in performing it dynamically or statically. The author continued to use dynamic method to change the PE header.

Chapter 3. VIRUS ATTRIBUTES

In this chapter, an exploration into how viruses, Trojans, and worms work will be investigated. Analysis is needed to determine what system API calls should be stopped and what combination of API calls will be an indication of malicious code. Malicious code has some common threads, which will be explored in the following chapter.

Escalating privileges

Services

The creators of malicious code often write their viruses to register themselves as services to Microsoft Windows. A service is automatically started when Microsoft Windows boots and is spawned from a privileged process. Since most spawned processes inherit the security descriptor from the parent process, services run in the privileged mode in Windows (Brown, 2000). This privileged mode has the full rights of the operating system. Malicious code loves this kind of access. Anything that is running in system level context will not be registered to the user. The user cannot see the process running. To put this in another way, the user cannot press the alt-control-delete combination to see this program.

Local buffer Overflows

Buffer overflows are often used to escalate privileges. An attacker would want to overflow a privileged process to escalate his or her reduced privileges to that of the overflowed program. This is difficult to stop. Any one of the API functions could contain a buffer overflow problem. One would need to monitor every API call and develop a string-scanning algorithm to identify the buffer overflow attempt. The author believes that intercepting every single API call could put too

much strain on the system. As the author has witnessed, the system usually crashes when monitoring a large number of API calls.

Enumeration

Rogue code often checks to see if other programs, such as anti-virus, are running. These viruses also use the enumeration of processes to figure out to which running processes to attach themselves. The API call is EnumProcesses() and can be called from a legitimate program checking to see if a previous version of the software is running.

Registry

The system registry is an important arena in which to stop malicious code. Many known viruses and Trojans like to hide in the registry and for good reason. The registry is a vast repository of information that is crucial to Microsoft products. It contains the majority of all the state information for the operating system. Stopping registry manipulation is one of the main goals of this thesis. An obvious problem arises; one cannot completely shut down the registry because normal programs will need access to this information. The solution is to block the most commonly attacked keys. A normal program would not normally access these keys, but malicious programs frequently do. I have attempted to identify specific keys that are favorites to malicious code writers in an endeavor to impede the infecting process.

How the Registry Works

Microsoft implemented a registry file with the introduction of Windows 95. The need for a storage house of state information was brought on by the fact that previous to Windows 95, information was stored in an .ini file. These files were simple text files and afforded little or no

security. With the implementation of a registry, security was added so that state information was not as easily accessible.

The registry has a tree like structure and is similar to the directory structure that is common to file systems. There are four main trees, HKEY_LOCAL_MACHINE, HKEY_CLASSES_ROOT, HKEY_USERS, and HKEY_CURRENT_USER. These are always open according to the Microsoft Platform SDK and can be accessed with calls through the API. This paper will now introduce keys and sub-keys, which can be quite confusing. A key is a directory structure that holds sub-keys. Sub-keys contain important state information for a program. Here is an example:

Key = SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce

Sub-Key =BearShare

The sub-key BearShare, which is actually a program that starts up on the author's computer, has a data type of REG_SZ and has the data of D:\PROGRA~1\BEARSH~1\BEARSH~1.EXE /m. This key allows the program BearShare to start when a computer boots up. The above example shows a key that will be monitored by this security program.

Writing Files

Files are more complicated than registry access. There are legitimate reasons why programs would want to overwrite a file in the Windows system directory. A downloaded program might need to overwrite an old DLL file that resides in the Windows system area. The

user should know what kind of file has been downloaded and should have a reasonable expectation of what directories are going to change from these downloaded files.

Watching file usage is an important aspect of identifying malicious code activity. A program can be categorized based upon the type of files that it writes. An example would be a program that iterates through all the folders and subfolders on a hard drive. Very few programs need to iterate through the entire directory structure. Only backup programs or special programs like disk defragmenters need this type of access.

Files and Directories Viruses like to Change

What follows is a list of commonly edited files and directories that are attacked by malicious code:

System.ini – could change the shell equals line to boot a program

Wininet.ini - Windows uses this file to install and uninstall programs on reboot

Autoexec.bat - As mentioned before, this file can be used to start programs up automatically when the system boots

Config.sys - Malicious code can change this file to start up software drivers

Win.ini - Contains load and run lines that could be used to start malicious code as well as a shell line that is vulnerable to attack.

These directories are commonly attacked:

System root - The system root directory is a favorite location for many programs to hide. Usually malicious code will write to this directory and give the new file a clever name so that a user doesn't suspect trouble.

Root directory - This is where autoexec.bat and config.sys as well as some critical startup files like boot.ini are stored

System temp directory - Not as critical as the other directories listed here but still the author would like to warn the user of access to this directory

Recycle bin - Virus writers figured that the user would not notice new files in the recycle bin and thus it is another favorite of malicious code writers

Startup folders - Places where code is automatically executed when Windows starts up.

Email Access

Most malicious code implementations are spread by email. Most of the malicious code that accesses emails uses MAPI to interface with the Outlook client to read the contact list. It is the hackers' belief that their code has a better chance of propagating if the virus is sent out through trusted users. This belief exploits a trust relationship. A virus writer can find these trust relationships in the contact list in outlook. The user will have the option of blocking MAPI calls to outlook.

Internet Access

Microsoft has many different ways of accessing the network. This is the most difficult attribute to stop due to the magnitude of ways to access the network through the API. This

security program makes no attempt to stop access either to or from the internet. The author believes that this is an important operation to stop; however, the complication of doing so was overwhelming.

Chapter 4. IMPLEMENTATION

This chapter covers the implementation of the security program. In the previous chapter, malicious code attributes were explained. This chapter discusses the actual API calls that will be used to block malicious code. After the API calls are discussed, a brief explanation on how the security program works is presented to the reader. The following diagram illustrates what the security program's front end looks like.

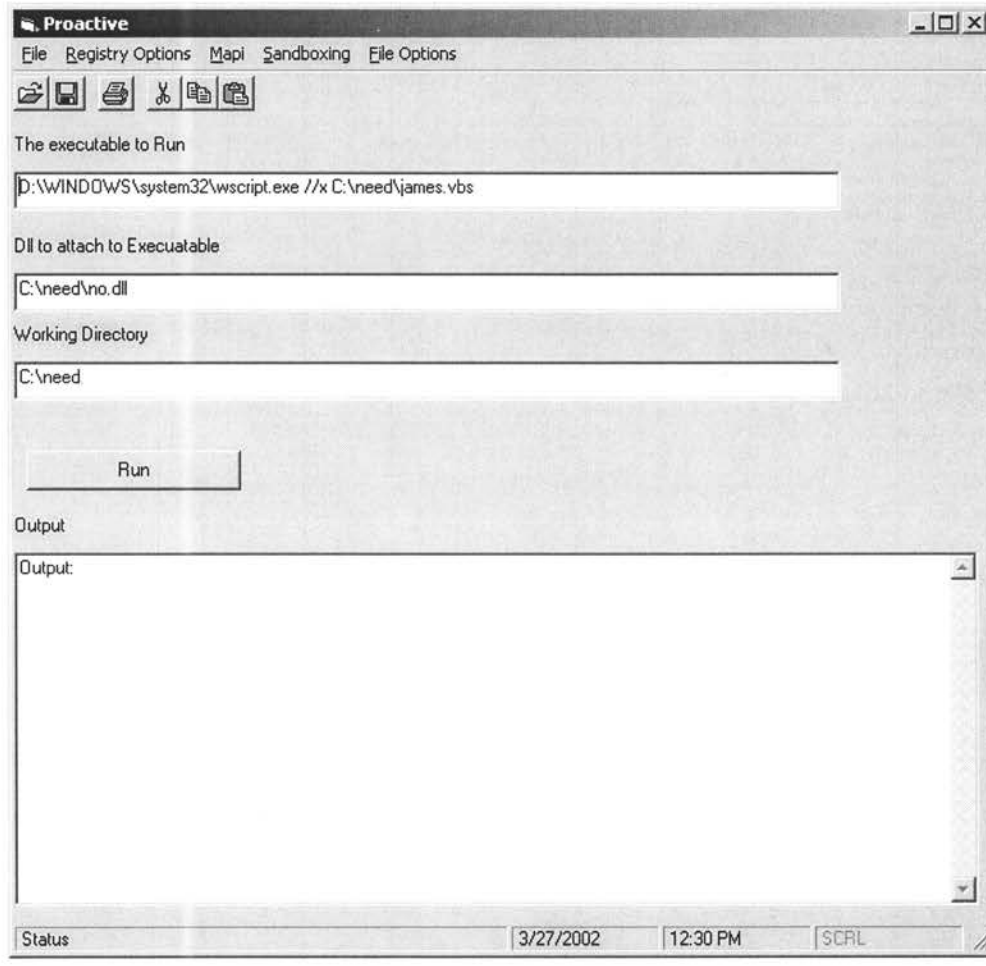

Figure 9. The Front End

Now let's look at the actual API call monitored.

Registry API Calls

Table 1. Monitored API Calls

Name	Description
ReqQueryValueExA	By far the most important call, this function allows a program to get a system handle to a specific key
RegOpenKeyA	Open the sub key which is under the specific key open with ReqQueryValueExA
RegSetValueExA	Sets the type of the data as well as the data in a sub key
RegEnumValueA	Enumerate through sub keys in a key
RegOpenKeyExA	Opens the key

Now that the API calls have been specified, one must determine what access to the registry is allowed and what is denied. The following is a list of keys that have been added to a text

file. These keys are placed on the restricted list because they are favorites for malicious code writers and most ordinary programs do not access these keys.

NeverShowExt - This key is used to never show an extension. This feature can be used to trick users into running something they didn't intend to. There is no need for a legitimate program to change this key.

The run keys - There are a couple of sub keys that allow programs to automatically execute when the operating system starts up (Chirillo, 2001). Most programs do no write to these keys but a couple of them, like file sharing programs, do.

Here are the run locations that are monitored:

- Hkey_Local_Machine\Software\Microsoft\Windows\CurrentVersion\RunServices
- Hkey_Local_Machine\Software\Microsoft\Windows\CurrentVersion\RunOnce
- Hkey_Local_Machine\Software\Microsoft\Windows\CurrentVersion\RunOnceEx
- Hkey_Local_Machine\Software\Microsoft\Windows\CurrentVersion\Run
- Hkey_Local_Machine\Software\Microsoft\WindowsNT\CurrentVersion\Winlogon

- Hkey_Local_Machine\Software\Microsoft\Windows\CurrentVersion\Run
- Hkey_Classes_Root\exefile\shell\open\command

Here is a list of keys that viruses change so that they can launch when a user executes an .exe file:

- Hkey_Classes_Root\exefile\shell\open\command\
- Hkey_Local_Machine\Software\Classes\exefile\shell\open\command

There are many more keys that users might want to block. It is very possible that a user might want to add the registry keys that control file associations. These keys are another probable area of attack. This security program reads in from a user defined text file containing keys that one wishes to block. These keys are then sent through a named pipe to the security DLL. Remember, this DLL is in the process of the software one wishes to monitor. For the security program to run, the user must select a text file with registry keys as show in the illustration.

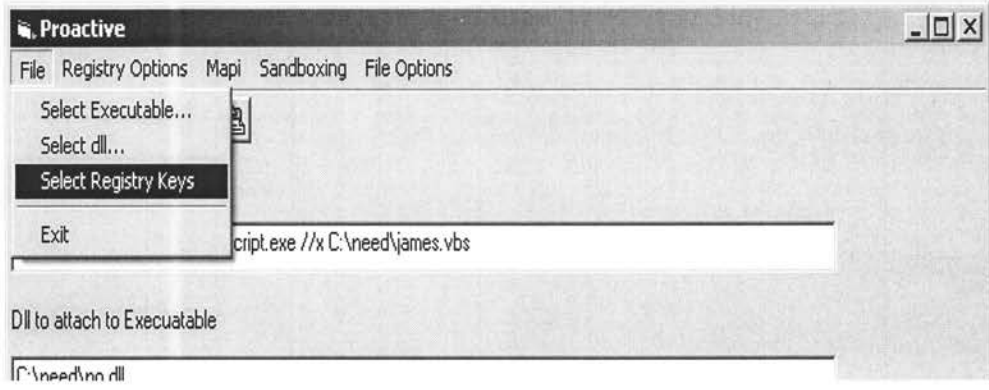

Figure 10. Selecting the Registry File

File System Calls

Now that the registry has been covered one can turn attention to file management. This security program must concentrate on the files accessed as well as the files being created or deleted. What follows is a list of specific API functions that will be monitored.

File API Calls

Table 2. Windows File Application Protocol Interface

Name	Description
`OpenFile`	Function used to create, open, reopen, or delete a file.

`MoveFileExW`
`MoveFileExA`
`MoveFileA`
`MoveFileW`
	Wide and ANSI definitions of move file. Attacker could attempt to move a file into a directory to subvert this security program.

`WriteFile`	The WriteFile function writes data to a file and is designed for both synchronous and asynchronous operation.

Table 2. Continued

`CopyFileExA` `CopyFileExW`	Function to copy an existing file or a directory, including its children, to a new directory.
`GetFileAttributesW`	Function retrieves attributes for a specified file or directory, also used to walk through a directory
`CreateDirectoryExW` `CreateDirectoryW`	Creates a new directory with security descriptors.
`DeleteFileW` `DeleteFileA`	Deletes a file from the system.
`CreateFileW`	Creates a new file on the system.

The user has an option with this security program to use sandboxing. Sandboxing restricts all of these API calls to two directories. The first directory is the working directory that is user supplied. The second directory is built into the DLL. This directory is the Windows system directory. In addition to sandboxing, the user also has the option of disabling all file creation and deletion. The option to stop file creation and deletion occurs in the working directory as well as

the system directory. With these three options enabled, the piece of code that is being monitored by the security program should be unable to write, read, or delete a file.

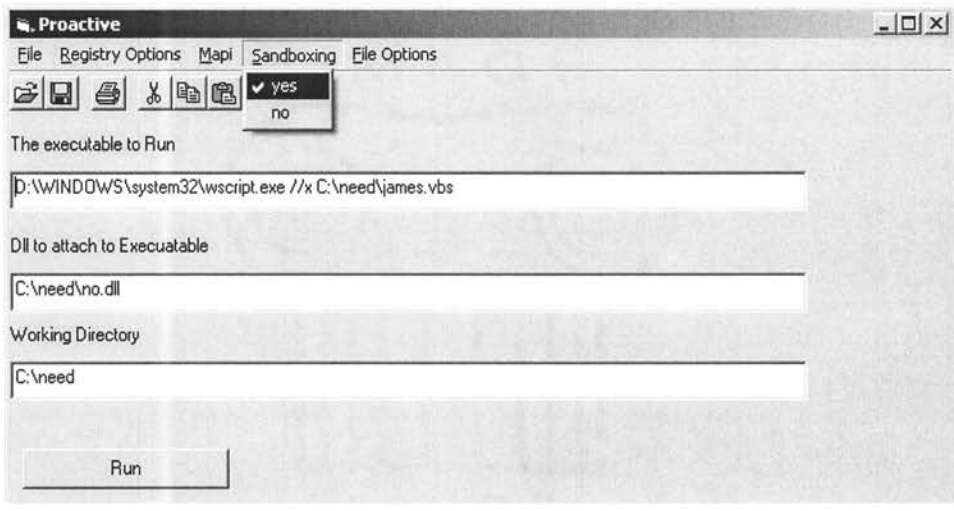

Figure 11. Menu Options

Overview of the Security Program

The operation of the security program is quite simple. The user selects options in the menu bar to determine how tightly to restrict the program to be monitored. The user then selects the executable that he or she wishes to run. If the executable selected has a .vbs extension, the security program changes the executable line to facilitate the script file. Then the user selects the DLL that will be attached to the executable. This DLL is usually no.dll but for expansion purposes this option was added to allow other DLLs to be used. The file with the registry keys is then selected. The user is now ready to execute the program. The front end of the program sends all this configuration data as well as the security keys over to the DLL that is running in the specified program. The DLL will report back to the front end when any blocked action occurs.

The status bar at the bottom of the program tells one the current state. When the executable that is being monitored is done, the DLL is unloaded and the security program is ready to run another program.

Difference between executables and scripts

There is a difference between .exe files and scripts. An .exe file is a stand-alone executable file that has been compiled to executable code. It can be brought to many different windows machines and will run on all of them. This is the target of the security program outlined by this thesis.

The security program injects itself by rewriting the PE header and loading the custom DLL. This method could not be applied to document files or scripts. These types of files might contain API calls but without a PE header to change, the author was unable to architect a way of loading a custom DLL to monitor for malicious code. However, other security products monitored not only executables as well as scripts, active X components, and java scripts. The author believes these programs used a simple method to monitor non-compiled code. One simply monitors the programs that run the scripts or word documents. The scripts, active X components and java code must be interpreted to machine language by an executable program. Using this knowledge, the author was able to test the security program with the "I Love You" virus, which is not an executable but a script.

Technical Description

In this section, the author will describe the technical problems that were solved in order to produce this security project.

Named Pipe

One of the first problems was to decide if a graphical user interface was needed for this project. This program could have just encompassed the security DLL. One possible solution was to compile ten differently configured security DLLs. Each DLL would be unique, offering an array of protection to the user. For example, one DLL would have closed down file access where another would have limited the access to the registry. This idea was quite simple since the DLL would have been static. The user simply picked the right security DLL for the access he or she wanted the downloaded program to be run under. This idea was rejected by the author, however, as it did not allow the user to select all possible DLL combinations. The next possible solution explored was the idea of a user front end. In this solution, the user could select the exact options that he or she desired. Implementation of this solution presented a unique problem: How would the front end talk to the security DLL? The answer came in the form of a named pipe connection between the DLL and front end (Walnum, 2000). A named pipe works exactly how one would expect. If you put data in one end, it comes out the other. There were some initial timing difficulties including when to start the named pipe in the DLL, which were resolved by using the timeout features available in the programming language (Brain, 2001).

Data Types

Another constantly occurring problem was in the conversion of data types used by Visual Basic to those used by C++. Strings in Visual Basic contain headers as well as size information. Before information could be sent across the named pipe, the Visual Basic string must be converted to a form the C++ could understand. Additionally, the parameters passed to most of these functions are wide string definitions of the form LPCWSTR. These are used for Unicode support.

As the author chose to block Unicode as well as ASCII functions, the Unicode must be stripped down to a comparable version for processing. The author wrote some comparative functions to facilitate this.

Threads

At random times, the security DLL would send information to the front end of the security program. This information must be read from the front end and then displayed to the user. What was needed was a dual natured Visual Basic program capable of continually reading input and processing user information. Unfortunately, Visual Basic does not have the capability to write multithreaded applications. The solution presented itself in the form of the Visual Basic timer. Visual Basic has the capability to create a timer mechanism to execute code at regular intervals. If the intervals were too short, the program would lose responsiveness. Conversely, if the intervals were too long, the named pipe would overflow with data. 100 millisecond time intervals were chosen and seem to work well. Although a thread would have been the ideal solution, the timer is adequate.

Determining API Calls

There was an unexpected problem in determining the proper API calls. The author was under the impression that the MSDN documented Win32 API calls were correctly named and could be used in intercepting API calls with the detours library. The author struggled to intercept the first API call and a look at the detours documentation library did not provide guidance. The author tried to create a file using the MSDN library command to create a file: createfile(). Upon first inspection, the author believed this was the correct syntax to use for interception. It was not until the author carefully examined the system DLLs, however, that he found a reference to

CreatefileW. With this as the function prototype for the detours library, the interception took place and he was able to intercept his first call.

Chapter 5. CASE STUDY, THE 'I LOVE YOU' VIRUS

In this chapter, the new security program will be put to the test with the "I Love You" virus launched in May 2000. The "I Love You" Virus is an ideal case study and test model for the following reasons. First and foremost, the source code to the worm was available online, making testing more complete. With source code, the virus can be debugged line by line which provides the author with a clear view of the operations performed by the virus. Secondly, the "I Love You" virus is relatively simple with no more then 300 lines of code. Third, this virus was one of the most costly viruses written, which some have speculated to cost billions of dollars. Finally, this virus was not an executable but simply a script. This allowed testing of the new security program in a situation where a legitimate scripting host, which will extensively use API calls to facilitate itself as well as the virus, is running. The new security program must let the scripting host run even while the virus code tries to execute, or more simply, it must distinguish between good and bad code. The next section explains the background of the "I Love You" virus and then a report card will be presented to see how well the new security program performed.

Background on 'I Love You'

The "I Love You" virus is a Visual Basic script that will run on Microsoft Windows systems that the scripting host is enabled. The virus arrives as an email attachment. Once a user executes the attachment, the virus goes to work. The payload of the virus overwrites specific files on ones hard drive and then sends out an infected email to everyone in the outlook address book.

In Detail

The virus begins by creating a copy of itself and storing it in a global variable. It then uses this global variable to make three files in the system, temp, and windows directories. Since this new security program has sandboxing technology, these writes are completely blocked. The "I Love You" virus is coded to continue on error, therefore the virus continues and tries to write to the registry. The "I Love You" virus follows the same pattern as many viruses. It attempts to add an entry into the

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run\

MSKernel32 key. The value written executes the virus when Windows boots up. This key is in the list of blocked keys so this call also becomes intercepted and blocked.

The virus continues with the modification of the Internet Explorer start page. It facilitates this by changing the registry entry for the start page. This action is also denied by the new security program when the option to block new keys is selected. The virus then enumerates through the directory structure scanning for files with a .vbs ending. If it finds one of these files, it changes the script file to have a copy of the virus. This is also blocked if the script file resides outside the current directory. If the current folder contains scripts, however, these files will be infected by the virus. This action can also be stopped by not allowing file creation, which is an additional option on the menu bar.

As the virus proceeds, it attempts to delete files with the .jpg or .jpeg extensions. All file deletions are blocked so this action fails. It then checks for .mpeg or .mp3 extensions and tries to create a new file with the same name as the music file. This new file contains a copy of the virus.

Once again, if these files are outside the directory structure in which the virus is running, this action will be blocked. Also if blocking file creation is enabled, the virus cannot create a file even in the running directory.

The virus's next stage is to look for an IRC client. If one is found, the "I Love You" virus attempts to create a new script.ini in the IRC directory. This action will be blocked if it is outside the virus's working directory.

Email is the next target for the "I Love You" virus. It attempts to access the infected user's outlook address book. This is facilitated through a function call to the MAPI services, which can be blocked by this security program. The virus receives no email addresses from this action and since it has no emails to send, it proceeds without sending infected emails to everyone in the outlook contact list.

Report Card

The new security program preformed admirably, stopping most, if not all the dangerous payload of this worm. No files were deleted. The virus does not start up with the booting of Windows and infected email was not sent. The only downfall of the new security program occurs when the virus is run in the same directory as music files, picture files and IRC files. This problem only occurs if one has not selected to block file create and file delete.

Chapter 6. POSSIBLE ATTACKS

In this section, an exploration into defeating this proactive security program will be performed. Through further research, it has become evident that there are shortcomings and possible security holes to this type of program. For these shortcomings to be exploited, however, the malicious code writer would need knowledge of the inner workings of this security software. Three possible ways the security program could be bypassed are as follows: Native API, direct jumps, and self-modifying code. It is possible to implement solutions for each of these obstacles; however, they are not presented in this work.

Native API

The first and conceptually the most difficult to defeat is the native API. In figure 2, a model was given to aid the reader in conceptualization of this security program. Although an accurate representation of the big picture, taking a look at how Windows implements the native APIs creates a more detailed picture.

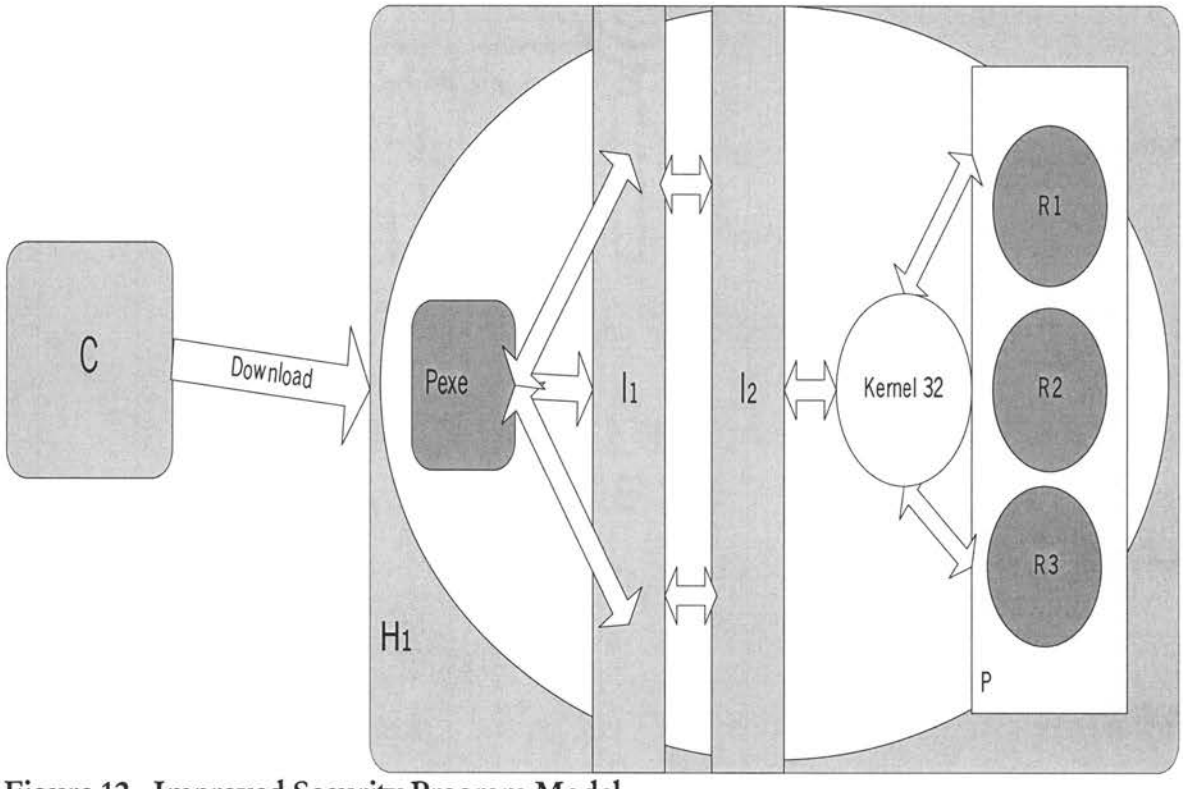

Figure 12. Improved Security Program Model

The model now includes two levels of application interface. I1 will now be defined as the user API and I2 is the Native API. The OS Kernel is added to the picture and is the object that directs undocumented native API calls to undocumented executive calls.

The Windows native API is vastly undocumented (Russinovich, 1998). Only a small amount of documentation can be found in the Windows Device Driver Kit. Even though Microsoft has not released documentation on these functions, programmers have figured how to use these functions with great accuracy. The native system APIs reside in a file called Nt.dll. The following figure shows the path an API call takes in a Windows system.

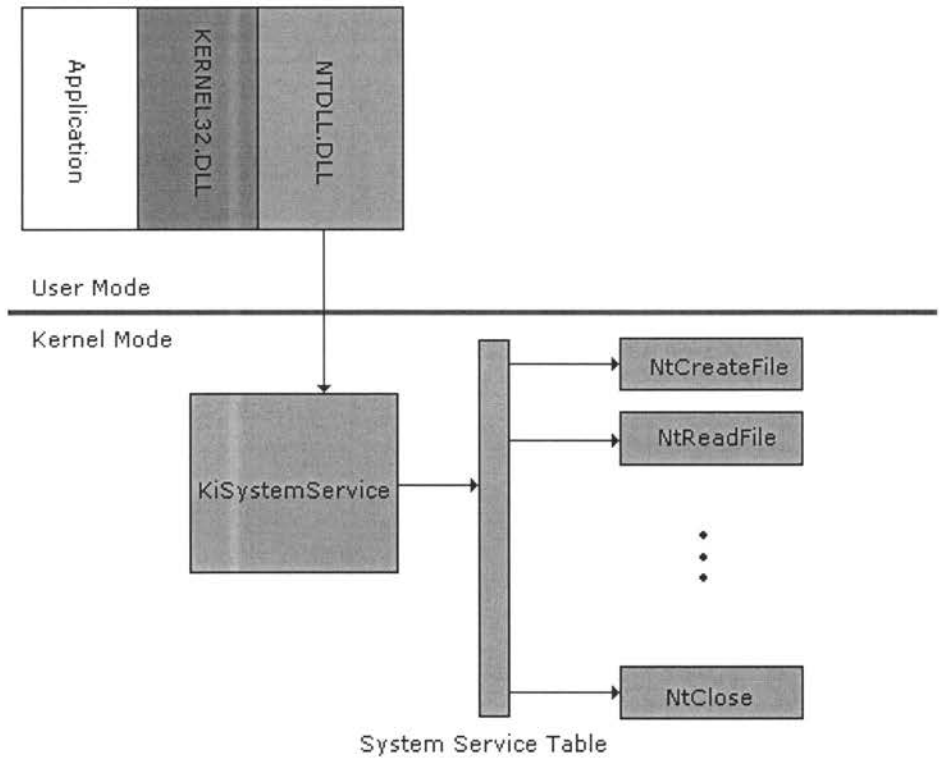

Figure 13. Execution of a Win32 API Call (adapted from Russinovich, 1998)

The question arises as to why the author chose to intercept user API calls and not native API calls. The answer to this question is dictated by Microsoft's documentation. As described in the detours literature, one must know the exact function definitions to use binary rewriting technology to monitor the API (Hunt, 1998). Since Microsoft has not documented this layer, the author believes it would have been nearly impossible to implement the detours library in this level.

The situation gets even more convoluted due to the fact that native applications exist that are not clients of the operating system. These native applications cannot be clients because they are used before the Windows 32 operating environment is initialized. The native applications bypass the user API and therefore would completely circumvent this security program. As of this

writing, the author is not aware of malicious code that is compiled in this manner. However, as proactive systems become more deployed, virus writers may turn to this method in order to bypass this type of security.

A solution to the problems posed by native API calls is presented through static analyzation. It would be possible to look at the executable in memory and determine if it makes these types of calls. Once this determination is made, the decision would be up to the user whether or not to proceed with this type of code.

Self-Modifying Code

Another issue is presented by self-modifying code. Self-modifying code is code that is capable of writing to itself and modifying its original intent. The consequences of this ability seem intuitive. A security program could scan an executable for API calls. The security program would detect no API calls and thus believe the program to be benign in nature. The executable is then launched and deletes all files on the hard drive. Attacks of this type can easily occur, as seemingly benign programs are able to morph into dangerous pieces of software during execution. Although this security program doesn't protect from this, there are methods in stopping self-modifying code. On possible way is to make the code segment of the executable read-only, which can be achieved by virtual memory protection.

Indirect Jumps

There exists another possible way of sidestepping this security program. Indirect jumps can transfer control to a location in memory at run-time (Twyman, 1999). Virus writers could use indirect jumps to transfer execution to native API calls. These locations can be outside the

programs memory space. Static analyzation cannot be used to determine indirect jumps since they occur at run time. To combat this threat, software fault isolation (SFI) can be implemented on this executable. SFI uses binary rewriting technology to scan the executable and place run time checks on read, write, and jump operations. This is accomplished by using segment matching which compares the high bits of these jump addresses to the correct memory segment. If the high bits match, the jump is safe. Conversely, if the bits do not match, an exception handler is run (Twyman, 1999).

Chapter 7. CONCLUSION

The author considers this security program to be a success. Its purpose was to demonstrate that API monitoring could be used as an effective measure against malicious code. This was successfully demonstrated with the "I love you virus".

Future Work

There are some remaining issues that the author did not explore due to time restraints. There is a question of the effectiveness of the security program when the virus is encrypted. Since the detours library performs binary rewriting technology in memory, it may be possible that an executable could be constructed to evade this technology. The solution to this problem is to let the virus decrypt itself and then attach the DLL to the virus in order to block its access. This is done by writing a piece of software to first look for decoding loops and then to later attach the DLL. Decoding loops are well understood and there is software known to the security industry for detecting these types of loops.

Another problem with this security program could occur if a virus was written to manipulate data without using the system API. This virus could be coded by writing very low-level assembly. Here is an example: A virus writer could write a low level piece of software to send out a packet on an Ethernet card without using Windows API calls.

Finally, there is a problem with undocumented window calls. If a virus writer was to define an undocumented call to perform an action like file deletion, this program would not stop it. Also, there are many different ways of deleting a file without actually deleting it. Instead of calling the API call to delete a file, which would be blocked by my program, a malicious code

writer could open the file and modify it to contain zero bytes of data. Another example would be moving the file to the recycle bin and letting the user delete it accidentally. Microsoft has thousands of different APIs. Understanding how they all come together is a difficult task. The author has no doubt that some API calls have been missed and these missed API calls could be dangerous to system integrity and should be blocked. Due to the fact that the author does not have complete understanding of Microsoft's API structure some calls slipped through the cracks of this security program.

Further work on this project would have to examine these aforementioned problems. In addition, this program should be expanded to block internet access. This is a daunting task but the author believes that it can be done. Blocking internet backdoors would be extremely helpful with stopping Trojan horses. File manipulation also needs to be added to this project. Viruses love to attach to programs and imbed themselves into these files. API calls could be monitored to stop this type of aggression against a system.

Final Thoughts

This solution is not the end of malicious code attacks. Like any program of this nature, hackers could subvert this program. If an attacker knows prior to writing a virus the exact API calls monitored, the attacker will be able to subvert this security program. The author believes this is the exact reason technical documentation on how other proactive solutions were implemented could not be found.

Overall, this program is simply another layer of security to be added into the security mix. It is the author's belief that the more layers added, the more difficult it will become for the

malicious code writers to prevail. This is another step to assist the information assurance community in completing their goal of a secure computer system.

BIBLIOGRAPHY

Brain, M.(2001). *Processes and Threads.* Win32 System Services, The Heart of Windows 98 and Windows 2000. Prentice Hall. Press, New Jersey.

Brown, K. (2000). Programming Windows Security, Addison-Wesley, New Jersey.

Chirillo, J. (2001). *Hacker Coding Fundamentals.* Hack Attacks Revealed, Wiley Computer Publishing, New York.

Clayton, W. (2000). *OS Core Programming.* Windows 2000 Programming Secrets, IDG Books, Foster City, CA.

Duzlevski, O. (1999). Microsoft Windows NT (In)security Model, *Hello, World!* Vol. 1 No. 2. Retrieved March 13, 2001 from: http://www2.latech.edu/~acm/HelloWorld.shtml.

Evans, D. and A. Twyman. (1999). Flexible Policy-Directed Code Safety. *IEEE Symposium on Security and Privacy*, Oakland, California, May 9-12, 1999.

Evans, D. (1999). Policy-Directed Code Safety. MIT PhD Thesis. October 19, 1999

Fraser, T., L. Badger and M. Feldman. (1999). Hardening COTS Software with Generic Software Wrappers. *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, Oakland, California, May 9-12, 1999.

Grimes, R. (2001). Malicious Mobile Code Virus Protection for Windows, O'Reilly and Associates, Sebastopol, CA.

Hunt, G. and D. Brubacher. (1999). Detours: Binary Interception of Win32 Functions. *Microsoft Research Technical Report*, MSR-TR-98-33.

Kaplan, Y (2000). API Spying Techniques for Windows 9x, NT and 2000. Retrieved March 2, 2002 from http://www.internals.com/articles_main.htm.

Ko, C. (2000). Logic Induction of Valid Behavior Specifications for Intrusion Detection. *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, Oakland, California, 14-17 May, 2000.

McKay, E. and M. Woodring. (2000). Strategies, Tools, and Techniques for Visual C++ Programmers, Addison-Wesley, New Jersey.

Nebbet, G. (2001). Windows NT/2000 Native API Reference. Retrieved March 20, 2002 from The Code Project database.

Pietrek, M. (1994). Peering Inside the PE: A tour of theWin32 Portable Executable File *Format. Microsoft Systems Journal.* Vol. 9 No.3.

Pietrek, M. (1994). Under the Hood. *Microsoft Systems Journal.* Vol 11. No. 12

Pietrek, M. (2002). Inside Windows, An In-depth look into the Win32 Portable Executable File Format. *Microsoft Systems Journal.* Vol. 17 No. 2.

Pandey, R. B Hashii (1998). Providing Fine-Grained Access Control For Mobile ProgramsThrough Binary Editing TR-98-08. Accessed Feb 3, 2002. from: http://www.cs.ucdavis.edu/research/tech-reports/1998/CSE-98-8.pdf.

Russel, R. (2000). *Buffer Overflow.* Hack Proofing your Network Internet Tradecraft. Syngress Publishing, Rockland, MA.

Russinovich, M. (1998). Inside the Native API. SysInternals Freeware. Accessed Feb 16, 2002. from: http://www.sysinternals.com/ntw2k/info/ntdll.shtml.

Russinovich, M. (1998). Inside Native Applications. SysInternals Freeware. Accessed Feb 16, 2002. from: http://www.sysinternals.com/ntw2k/info/native.shtml.

Szor, P. (2000). Attacks on WIN32 – Part II. *Virus Bulletin Conference, 2000.* Accessed Feb 14, 2002. from: http://securityresponse.symantec.com/avcenter/reference/attack.on.win32.pdf.

Twyman, A. (1999). Flexible Code Safety for Win32. MIT MEng Thesis. May 21, 1999.